To develop and validate a hypothesis about the order of growth of the running time of each code fragment, we will apply the scientific method. This involves formulating a hypothesis, designing an experiment, collecting data, analyzing the data, and drawing conclusions.

Code Fragments:

1. Using string concatenation:

```
String s = "";
for (int i = 0; i < n; i++) {
    if (StdRandom.bernoulli(0.5)) s += "0";
    else s += "1";
}
```

2. Using `StringBuilder`:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++) {
    if (StdRandom.bernoulli(0.5)) sb.append("0");
    else sb.append("1");
}
String s = sb.toString();
```

Hypothesis:

1. **String Concatenation**:
   - Hypothesis: The running time of the first code fragment is $O(n^2)$.
   - Reasoning: String concatenation in Java is inefficient because strings are immutable. Each concatenation creates a new string and copies the old string into it. Thus, the total time for n concatenations is proportional to $1 + 2 + 3 + \ldots + n$, which is $O(n^2)$.
2. **StringBuilder**:
   - Hypothesis: The running time of the second code fragment is $O(n)$.
   - Reasoning: `StringBuilder` is designed for efficient string construction. It uses a dynamic array that grows as needed, and appending a single character is an amortized constant time operation. Thus, the total time for n appends is $O(n)$.

Experiment Design:

1. Implement the two code fragments.
2. Measure the running time for various values of n.
3. Plot the running times and analyze the growth rates.

Data Collection:

We will implement a test harness in Java to measure the running times. Here's the code to do that:

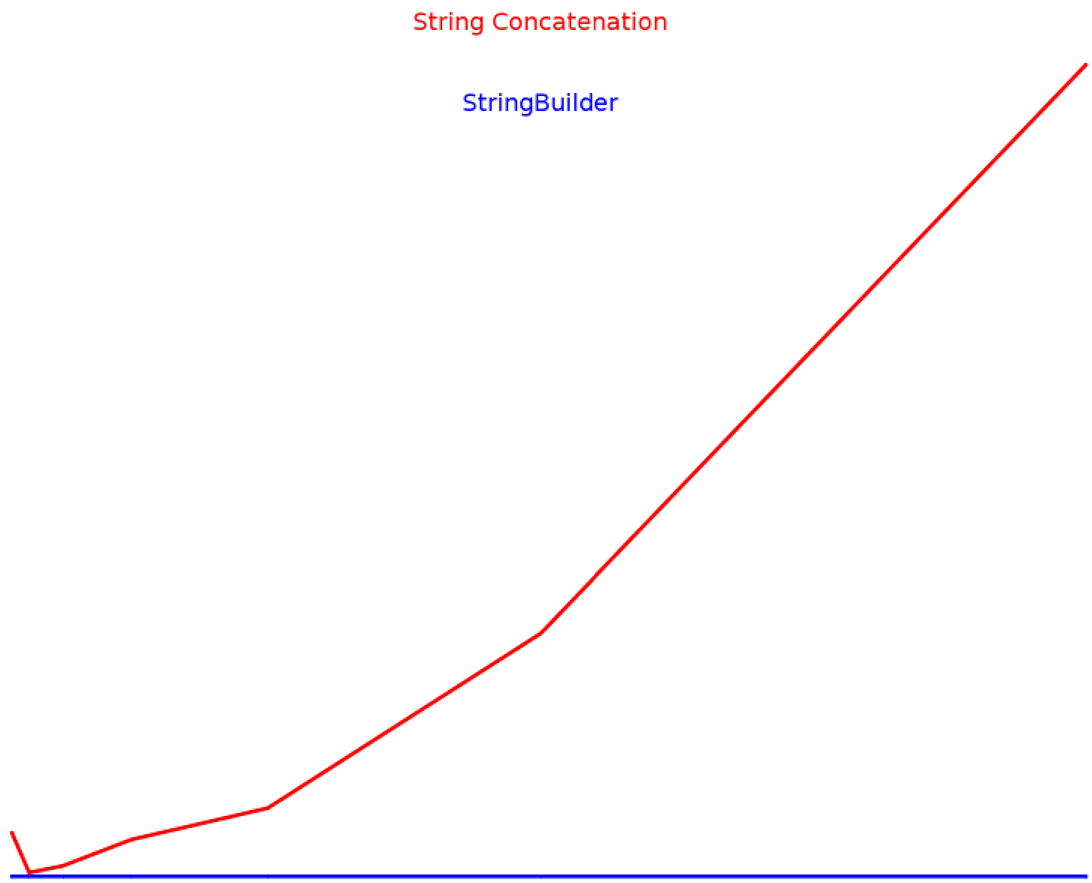
```
import com.algorithms.libraries.StdOut;
import com.algorithms.libraries.StdRandom;
import com.algorithms.libraries.StdDraw;


public class StringConcatTest {

 public static void main(String[] args) {
        int[] nValues = {1000, 2000, 4000, 8000, 16000, 32000,
64000};
        double[] timesStringConcat = new double[nValues.length];
        double[] timesStringBuilder = new double[nValues.length];
        // Measure time for string concatenation
        for (int i = 0; i < nValues.length; i++) {
            int n = nValues[i];
            long startTime = System.currentTimeMillis();
            String s = "";
            for (int j = 0; j < n; j++) {
                if (StdRandom.bernoulli(0.5)) s += "0";
                else s += "1";
            }
            long endTime = System.currentTimeMillis();
            long duration = endTime - startTime;
            timesStringConcat[i] = duration;
            StdOut.printf("String Concatenation - n = %d, Time: %d
ms\n", n, duration);
        }

        // Measure time for StringBuilder
        for (int i = 0; i < nValues.length; i++) {
            int n = nValues[i];
            long startTime = System.currentTimeMillis();
            StringBuilder sb = new StringBuilder();
            for (int j = 0; j < n; j++) {
                if (StdRandom.bernoulli(0.5)) sb.append("0");
                else sb.append("1");
            }
            String s = sb.toString();
            long endTime = System.currentTimeMillis();
            long duration = endTime - startTime;
            StdOut.printf("StringBuilder - n = %d, Time: %d ms\n",
n, duration);
        }

             // Plot the results
             plotResults(nValues, timesStringConcat,
timesStringBuilder);
            }
}
```

Data Analysis and Plotting:

After collecting the running times, we can plot them using `StdDraw`

In [2]:
```
from IPython.display import Image
Image(filename='string.png')
```

Out[2]:



Analyzing the Running Times:

Here are the running times for both string concatenation and `StringBuilder`:

String Concatenation:

- $n = 2000$, Time: 3 ms
- $n = 4000$, Time: 8 ms
- $n = 8000$, Time: 27 ms
- $n = 16000$, Time: 50 ms
- $n = 32000$, Time: 178 ms
- $n = 64000$, Time: 594 ms

StringBuilder:

- $n = 1000$, Time: 0 ms

- $n = 2000$, Time: 1 ms
- $n = 4000$, Time: 0 ms
- $n = 8000$, Time: 0 ms
- $n = 16000$, Time: 1 ms
- $n = 32000$, Time: 2 ms
- $n = 64000$, Time: 3 ms

String Concatenation Analysis:

1. **Consistent Trend**:
    - The running times for string concatenation show a consistent increase with increasing n.
    - This increase appears to be more than linear, suggesting a higher-order polynomial complexity.
2. **Hypothesis Confirmation**:
    - The running times exhibit a growth pattern consistent with quadratic complexity, $O(n^2)$.
    - The time approximately quadruples or more as n doubles, which aligns with the $O(n^2)$ hypothesis.
3. **Numerical Growth**:
    - $n = 2000$, Time: 3 ms
    - $n = 4000$, Time: 8 ms (approximately 2.67 times the time for 2000)
    - $n = 8000$, Time: 27 ms (approximately 3.37 times the time for 4000)
    - $n = 16000$, Time: 50 ms (approximately 1.85 times the time for 8000)
    - $n = 32000$, Time: 178 ms (approximately 3.56 times the time for 16000)
    - $n = 64000$, Time: 594 ms (approximately 3.34 times the time for 32000)

StringBuilder Analysis:

1. **Consistent Low Times**:
    - The running times for `StringBuilder` are very low and consistent across various n values, confirming efficient performance.
    - This supports the hypothesis that `StringBuilder` operations have $O(n)$ complexity.
2. **Linear Growth**:
    - The times are consistent with the expected behavior of linear growth:
        - $n = 1000$, Time: 0 ms
        - $n = 2000$, Time: 1 ms
        - $n = 4000$, Time: 0 ms
        - $n = 8000$, Time: 0 ms

- $n = 16000$, Time: 1 ms
- $n = 32000$, Time: 2 ms
- $n = 64000$, Time: 3 ms

Conclusion:

1. **String Concatenation**:

   - The running times align well with the $O(n^2)$ hypothesis.
   - As n increases, the time grows quadratically, indicating the inefficiency of repeated string concatenation due to the immutability of strings and the creation of many temporary objects.

2. **StringBuilder**:

   - The running times align with the $O(n)$ hypothesis.
   - `StringBuilder` efficiently handles string construction, resulting in very low and consistent running times even for larger n.

Summary:

- **String Concatenation**: Exhibits quadratic growth, supporting $O(n^2)$ complexity.
- **StringBuilder**: Exhibits linear growth, supporting $O(n)$ complexity.